Vertical Profiling of Java Applications through Aspect-Oriented Dynamic Binary Instrumentation

Jonas Maebe Dries Buytaert Lieven Eeckhout Koen De Bosschere
ELIS Department, Ghent University, Belgium
{jmaebe,dbuytaer,leeckhou,kdb}@elis.UGent.be

Abstract

Understanding Java application behavior is non-trivial given the virtual machine on which the Java application gets executed. This paper proposes vertical profiling through dynamic binary instrumentation for analyzing Java applications. The idea of vertical profiling is to provide a accurate picture of the program's execution that crosscuts various layers of the execution stack. Vertical communication between the virtual machine and the dynamic instrumentor is implemented through a callback mechanism which makes vertical profiling very easy to implement because very few modifications need to be made to both the virtual machine and the dynamic instrumentor. Our second contribution is aspect-oriented instrumentation (AOI) which is a natural way for expressing instrumentation, especially in case vertical instrumentation is the target. Finally, we build a vertical profiling framework called DJ and an associated VAOMI vertical aspect-oriented instrumentation language that allows for quickly building customized vertical profiling tools. We evaluate the overhead of vertical profiling and demonstrate its applicability through two applications: vertical cache profiling and object lifetime computation.

1. Introduction

Understanding the behavior of software is of primary importance to improve its performance. Application developers, system software designers, computer architects, etc. all need a good understanding of an application's behavior in order to optimize overall system performance. Analyzing the behavior of applications written in imperative languages such as C and C++ is a well understood problem. However, understanding the behavior of modern software that relies on a runtime system, which introduces an additional layer of virtualization, is much more challenging. This additional layer is typically called a virtual machine (VM). The popularity of virtualization software has grown significantly over the recent years with programming languages such as Java and .NET. The reasons for the increased popularity of virtualization environments are portability, security, robustness, automatic memory management, support for reflection, object-oriented programming, etc. Virtualization though makes the behavior of modern software hard to understand, or at least harder to understand than software written in imperative languages. When looking at the lowest level of the execution stack, *i.e.*, when looking at the individual instructions being executed on the host machine, it is hard to understand the application's behavior because of the fact that the virtualization software gets intermixed with application code at that level of the execution stack. However, when the goal is to understand the application's behavior, the lowest level of the execution stack really is the level to look at. For example, an application developer might be interested in finding the lines of code causing the highest cache miss rates; these can only be found by tracking native instructions. For application developers, virtualization makes it hard to understand the application's behavior since any performance metric measured at the lowest level in the execution stack includes both the application as well as the virtual machine.

In this paper we propose vertical profiling for analyzing Java applications. This is done by adding a dynamic binary instrumentation layer beneath the virtual machine that monitors the native instructions being executed. The instrumentation layer communicates with the virtual machine through a *callback* mechanism. The callback mechanism enables the virtual machine to inform the instrumentation layer on a number of issues, for example when an object is created, moved, or collected, or when a method gets compiled or re-compiled, etc. The information exchanged between the virtual machine and the instrumentation layer typically is a mapping of high level concepts such as methods, lines of code, objects, object types, etc. to native address ranges. By doing so, the instrumentation layer is capable of linking the low level behavior to high level constructs in the application, in the VM, etc., effectively collecting a vertical profile. An important advantage of employing binary instrumentation is that very few changes need to be made to the virtual machine in order to build a vertical profiling framework. Obtaining a similarly detailed vertical profile without binary instrumentation would have required many more changes in the VM. In addition, modifying the VM might perturbate the results being measured as the instrumentation code may change code and data layout. Dynamic binary instrumentation underneath the virtual machine alleviates this issue.

An additional requirement for a powerful instrumentation framework is an easy to use instrumentation specification medium. We present aspect-oriented instrumentation for this purpose. The idea behind aspect-oriented instrumentation (AOI) and aspect-oriented programming (AOP) is to specify functionality that concerns whole programs in a modular way. This is in sharp contrast to imperative instrumentation; aspect-oriented programming gives a much more natural way of expressing instrumentation. In this paper, we propose the *VAOMI (Vertical Aspect-Oriented Memory Instrumentation)* language, an aspect-oriented instrumentation language geared towards building customized vertical profiling tools for analyzing the memory behavior of programs written in object-oriented languages.

We demonstrate the feasibility of our vertical profiling proposal using aspect-oriented dynamic binary instrumentation by proposing the *DJ (DIOTA-Jikes RVM)* system, which combines the Jikes RVM [1] with the DIOTA instrumentation tool [10, 9]. We show that our DJ system is very effective and powerful for profiling Java applications. We discuss two diverse applications of the DJ system for analyzing the memory behavior of Java applications. In the first application, we study the cache behavior of Java applications. We show that DJ allows for tracking miss rates per object type and per method. This allows for a very powerful application analysis framework. For example, using the DJ vertical instrumentation framework it is easy to collect the top most cache miss causing lines of code, or the top most cache miss causing object types, etc. This is invaluable information for an application developer who wants to optimize his piece of software. In our second application, we show that the applicability of the DJ system goes beyond vertical profiling. Object lifetime is an important behavioral characteristic that is often used for analyzing the memory behavior of object-oriented programs. Computing an object's lifetime, although conceptually simple, is challenging in practice within a VM because the VM needs to be adjusted in numerous ways. This is both very time-consuming and error-prone. Computing object lifetime distributions using DJ only requires a few tens of lines of VAOMI code.

This paper makes the following contributions.

- We propose vertical profiling for analyzing Java application behavior. Vertical profiling allows for correlating low level information to high level information such as methods and objects. Our vertical profiling approach uses dynamic binary instrumentation for obtaining these vertical profiles. The key enabler for vertical profiling through dynamic binary instrumentation is a newly proposed callback mechanism which allows for efficiently communicating high level information from the virtual machine to the dynamic binary instrumentor. The important benefit is that very few changes need to be made to the virtual machine in order to build a vertical profiling framework.
- We propose aspect-oriented instrumentation as a way for specifying the desired vertical profile. For example, aspect-oriented instrumentation allows for easily tracking all objects of type T, or all memory locations accessed from method M, etc. Aspect-oriented instrumentation is a much more natural way of expression than todays imperative instrumentation tools; and this is especially convenient for specifying vertical profiles. We present the VAOMI language for specifying customized vertical instrumentation routines.
- We demonstrate our vertical profiling approach through aspect-oriented dynamic binary instrumentation in a Java application profiling framework, called DJ, that is built around the Jikes Research Virtual Machine and the DIOTA instrumentation tool. We present two applications that clearly show that the DJ environment along with the VAOMI language is a powerful Java instrumentation framework: vertical cache miss profiling and object lifetime computation.

The DJ vertical profiling environment will be made publicly available if the paper gets accepted.

2. Vertical profiling through dynamic binary instrumentation

The basic idea of vertical profiling is to measure metrics at various levels of the execution stack and to link those metrics across the various levels, *i.e.*, low level metrics are linked to high level concepts, and vice versa. Figure 1 illustrates how vertical profiling through dynamic binary instrumentation works for profiling a Java application. At the top of the execution stack we have a Java application that needs to be profiled. The Java application together with a number of Java libraries runs on top of a virtual machine. The virtual machine translates Java bytecode instructions into native instructions. The binary instrumentation tool resides beneath the virtual machine and tracks all native instructions executed by the virtual machine. The key point of vertical profiling is that the virtual machine informs the binary instrumentation tool whenever an object is created, moved, or deleted; or a method is compiled, or re-compiled; etc. The binary instrumentation tool then keeps track of these high level events and associates native addresses to each of those. As such, the binary instrumentation tool builds a vertical profile by tracking low and high level events and by linking them together. The binary instrumentation tool does not track system calls. System calls are handled directly by the operating system, and cannot be handled by the binary instrumentation tool. This is a limitation for all instrumentation environments, unless the operating system (and their system call implementations) would be instrumented as well.

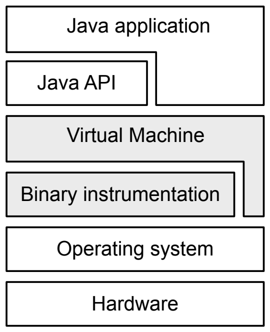

Figure 1. Vertical profiling of a Java application.

Implementing vertical profiling requires that both the virtual machine and the binary instrumentation communicate with each other. The following subsections discuss how this can be done in practice.

2.1 Virtual machine instrumentation

The virtual machine needs to notify the binary instrumentation tool upon the occurrence of various events. The events that are being tracked determine the power of the vertical profiling method. In our current setup the occurrences of the following events are tracked in the virtual machine and are communicated to the binary instrumentation tool:

- Class loading: When a new class is loaded and a new object type becomes available, the new class name is communicated to the binary instrumentor.
- Object allocation: When a new object is allocated, the object's type and memory location is communicated.
- Object relocation: When an object is moved by the garbage collector, the object's new location is communicated to the instrumentor.
- Method compilation: When a method is compiled, its name, memory location and a 'code to line number' map are communicated to the instrumentor.
- Method recompilation: When a method is recompiled, the method's location and 'code to line number' map are updated in the binary instrumentation tool.
- Method relocation: When code is moved by the garbage collector, the code's new location in memory is communicated.

- Memory being freed during garbage collection: when memory is freed, the address range of the free memory space is communicated to the binary instrumentor.

Note that this list of events being tracked by the virtual machine is just an example list of events that could be tracked during vertical profiling. Additional events could be defined and added to this list if desired. We found though that this list of events was sufficient for our purpose of analyzing the memory behavior of Java applications. These events already allow for a powerful vertical profiling method as will be shown in the remainder of this paper.

2.2 Dynamic binary instrumentation

A dynamic binary instrumentation tool takes as input a binary and an instrumentation specification. The binary is the program of interest; the instrumentation specification specifies what needs to be instrumented in the binary. The dynamic binary instrumentor then instruments the program of interest at run time. The binary instrumentation tool holds two copies of the same program under analysis. One copy holds the original binary; the other copy holds the instrumented binary. All data memory references in the instrumented binary refer to the original binary, *i.e.*, the instrumented binary sees the same data layout as the original binary. The instrumented binary is built as the program gets executed. Upon the first execution of a given piece of code, the instrumentor modifies the original code according to the instrumentation specification and stores the instrumented piece of code as part of the instrumented binary. By executing the instrumented binary, the desired instrumentation then is collected.

In order to be able to use dynamic binary instrumentation for analyzing the execution of Java applications, the dynamic binary instrumentation needs to be able to deal with self-modifying code. Todays virtual machines optimize and re-optimize hot code while executing an application. As such, the virtual machine frequently writes code in memory. The binary instrumentation tool that is tracking the virtual machine's execution needs to be able to accurately track this form of self-modifying code. The key problem with dynamic binary instrumentation and self-modifying code is that the code being executed is instrumented code and not the original code. If the original code would have written itself, then the instrumented code must produce the same result. The way self-modifying code can be implemented in a dynamic binary instrumentation environment is by marking pages that contain instrumented code as "read-only" using the virtual memory subsystem of the operating system. Subsequent writes to such pages then trigger protection faults that are intercepted and handled without the instrumented program noticing. The exception handler then flushes the instrumented code in the given memory page. Subsequently, writing to the memory page should be enabled temporarily and the instrumentor then needs to re-instrument the code.

2.3 Collecting a vertical profile

In order to support vertical profiling, the dynamic binary instrumentor needs to be able to handle events being tracked at the virtual machine level. This is done through so called *callbacks*, method calls generated by the virtual machine that are intercepted by the binary instrumentor—we assume for now that the binary instrumentor is dynamically linked to the virtual machine, which is the case in our setup as will be discussed later. A callback is a method called by the virtual machine, say method M with a number of arguments. Method M is a dummy method within the virtual machine. However, the dynamic binary instrumentor intercepts this method call and knows that a special action needs to be undertaken in case method M gets called by the virtual machine. In fact, the dynamic instrumentor replaces the execution of method M by the execution of method M* for updating the internal bookkeeping of the vertical instrumentor, *i.e.*, the internal bookkeeping related to objects, object types, methods, lines of code, classes being loaded, etc. Note that the arguments to method M can be read by the instrumentor by reading the arguments from the stack. This way, the callback in the virtual machine effectively triggers an event in the dynamic binary instrumentor. In order to implement vertical profiling we thus implement a number of callbacks to communicate information from the virtual machine to the binary instrumentor. For example, when allocating an object, the VM executes a callback `AllocateObject` with a number of arguments, namely the object type, its size and its memory address. And we have similar callbacks when a class is being loaded, when an object is being moved, or deleted, when a method is compiled or re-compiled, etc.

The dynamic binary instrumentor then keeps track of address ranges of the various methods and objects of interest and whenever a memory location is accessed, the binary instrumentor looks within its internal data structures for the method or object corresponding to the given address. The end result is that a vertical profile can be collected.

2.4 DJ implementation

The vertical profiling framework as discussed in the previous section is a general framework for vertical profiling. Any virtual machine could be employed in this framework and any dynamic binary instrumentation tool could be used as well. In our experimental framework, we use the Jikes RVM as our virtual machine and we use DIOTA as our dynamic binary instrumentation tool. We refer to our environment as the DJ (DIOTA-Jikes RVM) vertical profiling framework.

2.4.1 Jikes RVM

The Jikes Research Virtual Machine [1] is an open source Java virtual machine written almost entirely in Java. Jikes RVM uses a compilation-only scheme (no interpretation) for translating Java bytecodes to native machine instructions. In our experiments we use the *FastAdaptive* profile: all methods are initially compiled using a baseline compiler, and hot methods are recompiled using an optimizing compiler.

Modifying the Jikes RVM to enable vertical profiling was done very easily. We only had to insert around two hundred lines of code (including comments) into the virtual machine in order to trigger callbacks to the dynamic binary instrumentor. More specifically, we added a callback to the class loader, to the object allocator, to all garbage collectors when an object or code is being moved or deleted, and to all compilers and optimizers when a method is being compiled or optimized.

There is one particularity with instrumenting the virtual machine itself that needs special attention. Instrumentation cannot be activated until the virtual machine is 'ready'. This means that there are some virtual machine methods and objects that cannot be communicated to the binary instrumentation tool during virtual machine startup. This can be solved by communicating these virtual machine methods and objects as soon as the virtual machine is 'ready'. From then on, the instrumentor intercepts methods and objects during the program execution. This allows for vertically profiling the application as well as the VM.

Note that any virtual machine of interest could be used in a vertical profiling framework as long as the virtual machine can be augmented with callbacks which is not a problem for open-source and in-house virtual machines.

2.4.2 DIOTA

The dynamic binary instrumtation tool that we use in our DJ vertical profiling framework is DIOTA [10]. DIOTA stands for Dynamic Instrumentation, Optimization and Transformation of Applications and is a dynamic binary instrumentation framework for use on the Linux operating system running on x86-compatible processors. Its functionality includes intercepting memory operations, code execution, signals, system calls and functions based on their name or address, as well as the ability to instrument self-modifying code [9]. DIOTA is implemented as a dynamic shared library that can be hooked up to any program. The main library of DIOTA contains a generic dynamic binary instrumentation infrastructure. And this generic instrumentation framework can be used by so-called backends that specify the particular instrumentation of interest that needs to be done.

The general operation of DIOTA is very similar to that of other contemporary dynamic binary instrumentation frameworks such as PIN [8] and Valgrind [11]. All of these operate in a similar way as described in section 2.2. In fact, any of these dynamic binary instrumentation tools could be used for implementing a vertical profiling framework as long as self-modifying code is supported.

2.5 Discussion

2.5.1 Easy to setup

The key point of our vertical profiling approach is that only a limited number of callbacks need to be implemented in the virtual machine and supported in the binary instrumentor. And all the hard work of instrumenting low level events is done by the dynamic binary instrumentor. In addition, the dynamic binary instrumentor needs to keep track of the mapping between high level concepts such as methods, lines of code, objects, object types, etc. In fact, setting up a vertical profiling environment from scratch is done relatively easily. Our environment was set up within a few days of programming in order to get Jikes RVM and DIOTA cooperate in a vertical profiling environment.

2.5.2 Perturbation

When building a vertical profiling framework one has to be careful that the dynamic binary instrumentor does not instrument the VM code when the VM is computing the parameters for the callback methods. Dealing with this issue carefully allows for collecting highly accurate profiles, *i.e.*, the binary instrumentor only instruments pieces of code that would have been executed without any vertical profiling support embedded in the VM.

3. Aspect-Oriented Instrumentation (AOI)

Once a vertical profiling framework is setup, we also need an easy-to-use environment for building customized vertical profiling tools. This paper proposes a novel approach to binary instrumentation specification, namely aspect-oriented instrumentation (AOI), and more in particular we propose vertical aspect-oriented instrumentation which facilitates the construction of customized vertical profiling tools.

3.1 Aspect-Oriented Programming

Aspect-oriented programming (AOP) [6] is best known in the context of high level languages and software design methodologies, ranging from UML [16] and AspectJ for Java [5] to AspectC++ for C++ [14] to TinyC^2 for C [17]. The basic idea of aspect oriented programming originally came from the observation that not all functionality in a programming model can be cleanly separated into objects or modules. Some requirements crosscut entire class hierarchies, multiple modules and complete programs. Aspect-oriented programming allows for specifying a desired functionality that concerns the whole program in a modular implementation.

Logging an application's execution is one of the best known examples. Implementing a logging facility in a traditional manner without AOP requires that logging code is inserted in each and every piece of code. This is both very time-consuming, error-prone and in addition, hard to maintain from a software development point of view. AOP on the other hand allows for extracting this logging facility into a separate module, that is then *woven* by a *weaver* with the rest of the program at compile time or even at run time. AOP thus significantly improves software maintainability.

In general, an AOP language consists of *joinpoints*, *pointcuts* and finally the *advice*. A joinpoint specifies where and when one can interfere in the structure or execution of a program. This can range from source code line numbers to syntactical constructions to even run time events. A pointcut is a collection of joinpoints. Typically, a symbolic name can be associated with a pointcut for later reference. Finally, the advice is code that is associated with a pointcut. The advice will be executed whenever the conditions specified by the pointcut are fulfilled.

3.2 From AOP to AOI

The general idea of AOP languages of segregating crosscutting concerns in separate modules, is also very much applicable to the low level instrumentation of programs at the machine code level. In fact, instrumenting a binary involves inserting additional code across the entire program in order to measure a program metric of interest. And the instrumentation can be completely segregated from the original program. As such, aspect-oriented instrumentation (AOI) seems like a natural way for specifying binary instrumentation routines [12].

Nevertheless, low level instrumentation as done in all existing binary instrumentation tools such as ATOM [15], PIN [8], Valgrind [11], etc., all use an imperative way of specifying instrumentation code. These tools basically scan every instruction and instrument it according to the instrumentation specification. And this works perfectly fine for the purpose of low level instrumentation.

However, whenever vertical profiling is the focus, imperative instrumentation no longer is a satisfying approach. For example, tracking events concerning specific methods or objects is impractical to implement because the instrumentation specification (that needs to be programmed by the end user) then needs to scan all instructions in the binary in order to identify instructions belonging to the method of interest or to identify references to objects of interest, respectively. This is not a very efficient way of specifying instrumentation functionality. AOI on the other hand, allows for more easily specifying what actions need to be undertaken for specific methods, lines of code, objects, and object types. The code written under AOI does not require to scan all instructions to verify whether this instruction references an object but rather specifies what special actions (advices) need to be undertaken for a method of interest or an object of interest. As such, AOI is a much more natural way of expressing instrumentation functionality in the context of vertical profiling.

3.3 The VAOMI language

The VAOMI (Vertical Aspect-Oriented Memory Instrumentation) language that we propose in this paper is a domain-specific aspect-oriented language developed specifically for the purpose of vertical memory instrumentation in the context of an object-oriented language. It combines support for recognising individual memory accesses with the notion of high level concepts such as objects, object types, methods, lines of code, etc.

The grammar of the VAOMI language is displayed in Figure 2. A joinpoint that describes an event in the VAOMI language consists of a time qualifier followed by a memory event or an object event, followed by the advice code. The time qualifier specifies when the event should be triggered. This can be before or after the event of interest. The events that can be triggered are memory events or object events. For each of those, a number of parameters are given.

time_qualifier := `before` | `after`
params := `location_t const *` *loc* `,` `type_t const *` *type* `,` `void **` *userdata*
object_operation := `create` (params) | `copy` (params, params) | `delete` (params)
object_event := `object`:object_operation
memory_operation := `read` (params) | `write` (params) | `access` (params)
memory_operation_target := `object` | `nonobject` | `any`
memory_event := memory_operation_target : memory_operation
event := time_qualifier memory_event | object_event {*advice code*}

Figure 2. The grammar of the VAOMI language.

```
struct mem_access_t {
  int ip;            /* instruction pointer */
  int addr;          /* memory address being accessed*/
  int size;          /* number of bytes accessed */
  int ld_st;         /* load or store ? */
  int thread_ID;     /* thread ID */
}

struct location_t {
  struct mem_access_t *ma;  /* pointer to
                       mem_access_t structure */
  int method_ID;     /* method ID */
  char* method_name; /* method's name */
  int line_number;   /* line number in given method */
}

struct type_t {
  int type_ID;       /* object class ID */
  char* type_name;   /* object class name */
}
```

Figure 3. Data structures provided as parameters in VAOMI.

These parameters can then be used by the advice code. The advice code is the instrumentation code in C inserted by the end user.

3.3.1 Object events

An object event consists of the keyword `object` followed by an object operation. The object operation can be the creation (`create`), copying (`copy`) or deletion (`delete`) of an object.

3.3.2 Memory events

A memory event consists of memory operation target and the memory operation itself. The memory operation target can be an object, memory not belonging to an object or any of those. This allows the end user to focus the instrumentation of memory accesses to objects only, non-objects only, or to both objects and non-objects. The memory operation specifies the type of memory access that should be instrumented. This allows the end user to focus on reads, writes or both.

3.3.3 Parameters

The parameters that are provided by the VAOMI language are shown in Figure 3. These parameters can be used in the advice code for driving the instrumentation. The first parameter is a data structure that collects information concerning the 'location' of the object or memory event. This is done in the `location_t` structure. The first element in this structure is a pointer to a `mem_access_t` structure. This latter structure contains (i) the instruction pointer of the native instruction performing the object or memory operation, (ii) the object's memory location or in case of a memory operation, the memory location being accessed, (iii) the size of the object or in case of a memory operation, the number of bytes accessed in memory, (iv) whether this memory access is a load or store operation—note this has no meaning in case of an object operation, and finally (v) the thread ID of the thread performing the object or memory operation. The second and third element in the `location_t` data structure are the method ID and the method name performing the object or memory operation, respectively. The fourth and final element is the source code line number in the given method that corresponds to this object or memory operation.

The second parameter in the parameter list is a pointer to a data structure that specifies information concerning the 'type' of the object or memory operation. This `type_t` structure holds a type ID and a type name of the object or memory operation. This means that for every object being created, copied, deleted or accessed, the VAOMI language provides the end user with information concerning the object's type.

The third parameter in the parameter list (`void **`*userdata*) allows the end user for maintaining object-specific information. The end user may for example set up a data structure for a given object; the pointer to this data structure can be stored through this third parameter. The binary instrumentor then makes sure that this same pointer is available for all object and memory operations that refer to that same object.

3.3.4 VAOMI directives

The VAOMI language also comes with a number of directives that can be specified at the beginning of the instrumentation specification file. There are two directives in our current implementation, namely `#requires method_info` and `#requires object_info`. The purpose of these directives is to improve the performance, *i.e.*, to reduce the overhead, of the vertical instrumentation. The `#requires method_info` directive informs the dynamic binary instrumentation tool that the method ID, the method name and source code line number should be kept track of during binary instrumentation. The `#requires object_info` directive informs the dynamic binary instrumentation tool that the object type ID and the object type name should be kept track of. In a VAOMI instrumentation specification, the user can decide not to include any of these two directives, to include only one of these, or to include both directives. This will affect the amount of information that can be gathered during vertical profiling as well as the amount of overhead incurred due to the vertical profiling. For example, if a user is interested in measuring the cache miss rate per method and per source code line number, then there is no benefit in collecting per-object information. The user can then use the `#requires method_info` to disable tracking object-related information during the instrumentation run. This will limit the slowdown during vertical profiling.

3.4 VAOMI and DIOTA

In order to be able to use VAOMI instrumentation specifications with DIOTA, we also need a translator for converting the VAOMI statements as specified in Figure 2 into C-statements while keeping the advice code (that is written in C) untouched. The translated instrumentation specification is then linked with DIOTA and the Jikes RVM for running the vertical profiling.

```
after any:write (location_t const *loc, type_t const *type, void **userdata) {
  printf ("%p: W %p %d\n", loc->ma->ip, loc->ma->addr, loc->ma->size);
}
```

Figure 4. An example illustrating memory tracing instrumentation.

```
0:  #requires object_info

1:  after object:access (location_t const *loc, type_t const *type, void **userdata) {
      /* compute whether this object reference is a cache miss or not */
2:    hit = simulate_memory_access (loc->ma->addr, type->type_ID);
      /* update the per-type hit/miss information */
3:    update_per_type_miss_rate (type->type_ID, hit);
4:  }

5:  after nonobject:access (location_t const *loc, type_t const *type, void **userdata) {
      /* update the simulated cache content */
6:    simulate_memory_access (loc->ma->addr, -1);
7:  }
```

Figure 5. An example illustrating a per object-type cache miss rate vertical profiling specification.

3.5 Examples

We now illustrate the power of expressiveness of the VAOMI language through a couple of examples.

3.5.1 Example 1: Tracing memory writes

Our first example, see Figure 4, shows a memory tracing instrumentation specification. This instrumentation will capture all writes in the program code and will report the write's instruction pointer, memory address and size. As can be seen from this example, the VAOMI language is a very powerful binary instrumentation language. The expressiveness is high while the code itself is very intuitive. Compared to an imperative instrumentation specification languages, aspect-oriented instrumentation clearly is a more natural way of expression. Note that this example however does not illustrate the idea of vertical profiling. This is shown in our second example.

3.5.2 Example 2: Collecting object-type specific cache miss rates

The second example shown in Figure 5 illustrates the vertical profiling ability of the VAOMI language. The goal of this second instrumentation example is to collect cache miss rates per object type. This could be a useful application for a software developer when optimizing the memory behavior of a given application of interest. Line 0 specifies that the instrumentation needs to focus on per-object information only. Upon a memory access to an object (lines 1-4), the memory address is used by the cache simulator to update the cache's state and to update the type-specific data structure maintained by the instrumentor to keep track of the per-type miss rate. Memory accesses to other non-objects (lines 5-7) update the cache's state only and do not update per-type miss rate information because these memory references do not originate from object accesses.

4. Evaluation

Before presenting a number of applications for our vertical profiling framework, we first evaluate the overhead that comes from vertical profiling.

4.1 Experimental setup

We consider the SPECjvm98 benchmark suite[1], see Table 1, which is a client-side Java benchmark suite consisting of seven benchmarks. We run all SPECjvm98 benchmark with the largest input set (`-s100`). These benchmarks are run on the Jikes RVM using a 64MB heap and the generational mark-sweep GenMS garbage collector. And we run the DJ system on all of these benchmarks on a 2.8GHz Intel Pentium 4 system with a 512MB L2 cache and 1GB main memory. The operating system on which we run our experiments is Linux 2.6.10.

[1] `http://www.spec.org/jvm98/`

Benchmark	Description
jess	An expert shell system solving a set of puzzles
db	Makes database requests to a memory reisdent database
javac	Compiles Java to bytecode
mpegaudio	Decompresses MPEG-3 audio files
mtrt	A dual-threaded raytracer
jack	A Java source code parser generator

Table 1. The SPECjvm98 benchmarks used in this paper.

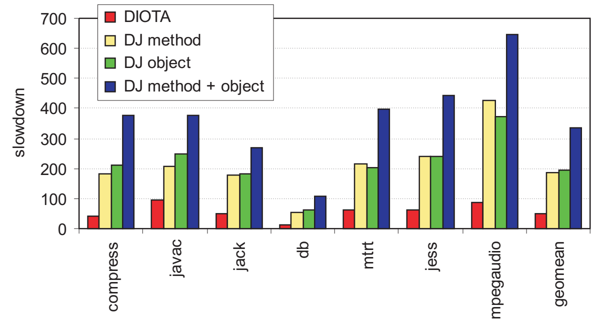

Figure 6. Slowdown due to vertical profiling.

4.2 Overhead from vertical profiling

In order to quantify the overhead of vertical profiling we consider the following scenarios.

- The first secnario measures the execution time for these benchmarks on the Jikes RVM without any dynamic binary instrumentation.
- The second scenario measures the execution time when running DIOTA underneath the Jikes RVM. In this scenario, DIOTA instruments all memory operations with an empty instrumentation routine. In addition, vertical profiling is disabled; none of the VAOMI directives was being set. This is to quantify the inherent overhead of dynamic binary instrumentation using DIOTA.

- The third scenario measures the execution time when running DIOTA underneath the Jikes RVM while enabling vertical profiling that only considers method-related information, *i.e.*, the VAOMI directive `#requires method_info` was set.
- The fourth scenario measures the execution time when enabling vertical profiling for measuring object-related information. The VAOMI directive `#requires object_info` was set.
- The fifth scenario measures the execution time for vertical profiling when both method-related and object-related information is kept track of. Both the `#requires method_info` and `#requires object_info` directives are set.

Figure 6 shows the slowdown for each of the above scenarios compared to the baseline scenario, *i.e.*, the first scenario. The inherent slowdown of dynamic binary instrumentation using DIOTA (scenario 2: 'DIOTA') varies from 11X to 94X. Recall this scenario does not incur any vertical profiling. The variation in slowdown from benchmark to benchmark comes from a number of sources: the number of indirect jumps, the amount of code being compiled and (re-)optimized by the virtual machine, etc. Comparing scenarios 3 thru 5 to scenario 2 quantifies the overhead due to vertical profiling. The average vertical profiling overhead varies between a factor 3.8X to 6.8X depending on what information is to be kept track of. Scenario 5 ('DIOTA method + object') quantifies the total overhead for VAOMI; and this incurs an average slowdown of a factor 6.8X. However, when making use of the VAOMI directives, see scenarios 3 'DIOTA method' and 4 'DIOTA object', significant reductions in overhead are obtained, *i.e.*, the average slowdown for scenarios 3 and 4 is only a factor 3.8X and 4X, respectively, compared to the 6.8X average slowdown for scenario 5. Note that these numbers were obtained from an implementation that was not optimized for speed; future work will study how to reduce the overhead.

5. Applications

We now discuss two example applications for vertical profiling through dynamic binary instrumentation.

5.1 Vertical cache simulation

The first application relates cache miss rates to high level concepts such as methods, source code lines, objects and object types. This could be invaluable information for software developers that are in the process of optimizing their code for memory performance. As is well known, the increasing gap in memory speed versus processor speed is an important problem in current computer systems. Poor memory behavior can severely affect overall performance. As such, it is very important to optimize memory performance as much as possible. Vertical profiling could be a very valuable tool for hinting the software developer where to focus on when optimizing the application's memory behavior.

Doing a vertical cache simulation requires that an instrumentation specification be written along the lines of the example discussed in section 3.5.2. In fact, the vertical instrumentation specification that we have written for this application extends the example from section 3.5.2 by also measuring cache miss information per method, per source code line, per object and per object type. We assume a 4-way set-associative 32KB 32-byte line L1 cache and an 8-way set-associative 1MB 128-byte line L2 cache. Both caches are write-back, write-allocate caches. The total instrumentation specification for this application was no more than 200 lines of code, including comments. The output of the profiling run is a table describing cache miss rates per method, per line of code, per object and per object type.

Selecting the per-method cache miss rates and sorting them by decreasing number of L2 misses results in Table 2. Likewise, selecting the per object-type miss rates and sorting them by decreasing number of L2 misses results in Table 3. The method names and object types specifications in these tables are given using the Java method and field descriptor notation[2]. In both tables we limit the number of methods and object types to the top five/three per benchmark in order not to overload the tables. The first column in each of both tables mentions the method or object type, respectively. The second column shows the percentage of memory references of the given method or object type as a percentage of the total number of memory references. The two rightmost columns show the number of L1 and L2 misses, respectively, along with the procentual miss rates. The software developer can use these tables to better understand the memory behavior of his application of interest and could even use this information for guiding memory optimizations at the source code level. For example, from Table 2 it is apparent that the `shell_sort` method in `db` is a method that suffers heavily from poor cache behavior. 20% of memory references in `db` occur within the `shell_sort` method. Of these memory references, 10.5% result in an L1 cache miss, and 31.7% of the L2 cache accesses are cache misses. As such, this method is definitely a method of concern to a software developer that strives at optimizing the memory performance of `db`. From Table 3 which shows per object type miss rates, we clearly observe that `db` suffers for a poor cache behavior. And this poor cache behavior seems to be apparent across a number of object types. For example, this table shows that the cache behavior for the `Vector` class is relatively poor with an L1 miss rate of 11.4% and an L2 miss rate of 38.6%. Note that our framework also allows for going even one step further, namely tracking down miss rates to individual objects. This would allow the software developer to really nail down the root cause of the poor memory behavior. We do not include an example of per-object miss rates in this paper because of space constraints, however, this could be easily done in DJ.

Since the `shell_sort` method in `db` seems to suffer the most from poor cache behavior, we therefore focus on that method now. Figure 4 shows the `shell_sort` method annotated with cache miss information, *i.e.*, L1 and L2 cache miss rates are annotated to each line of code. Line 13 seems to be the primary source for the high cache miss rate in the `shell_sort` method. The reason is that the `j+gap` index results in a fairly random access pattern into the 61KB `index` array. It's interesting to note that also Hauswirth *et al.* [4] identified the `shell_sort` method as a critical method for `db`.

5.2 Object lifetime

Our second example application is different from the previous application in the sense that it goes beyond vertical profiling. In fact, our second application is not vertical profiling at all, however, vertical profiling makes it very easy to compute. The purpose of this second application is to measure the distribution of object lifetimes. In this application we define the object lifetime as the number of memory accesses between the creation and the last use of an object. Knowing the allocation site and knowing where the object was last used can help a programmer to rewrite the code in order to reduce the memory consumption of the application or even improve overall performance [13]. Computing object lifetimes within a virtual machine is fairly complicated. First, the virtual machine needs to be extended in other to be able to store the per-object lifetime information. Second, one needs to be careful so that the computed lifetimes do not get perturbated by the instrumentation code. Finally, all object references need to be traced. This is far from trivial to implement. For example, referencing the object's header is required for accessing the Type Information Block (TIB) or vtable

[2] See `http://java.sun.com/docs/books/vmspec/2nd-edition/html/ClassFile.doc.html`

Method	Accesses	DL1 misses	DL2 misses
_201_compress			
`Compressor.compress()V`	45.4%	131324549 (7.7%)	553630 (0.4%)
`Decompressor.decompress()V`	45%	22492318 (1.3%)	504536 (2%)
`Input_Buffer.readbytes([BI)I`	1.5%	239804 (0.4%)	45149 (16%)
`Compressor.output(I)V`	3.1%	174822 (0.1%)	30962 (14.5%)
`Output_Buffer.putbyte(B)V`	0.4%	41291 (0.3%)	9365 (19.2%)
_213_javac			
`Assembler.add(IILjava/lang/Object;)V`	0.1%	147763 (2.4%)	35774 (17.1%)
`CompoundStatement.check(LEnvironment;LContext;JLjava/util/Hashtable;)J`	0%	214716 (14%)	31256 (11.4%)
`CompoundStatement.code(LEnvironment;LContext;LAssembler;)V`	0%	112464 (7.6%)	26769 (16.8%)
`Instruction.<init>(IILjava/lang/Object;)V`	0.1%	100472 (3.5%)	24400 (17.2%)
`MethodExpression.codeValue(LEnvironment;LContext;LAssembler;)V`	0.1%	135576 (5.6%)	22701 (11.9%)
_228_jack			
`TokenEngine.getNextTokenFromStream()LToken;`	0.9%	165627 (0.4%)	12395 (5.4%)
`JackConstants.printToken(LToken;Ljava/io/PrintStream;)V`	0%	20506 (1.1%)	4479 (12.4%)
`Token.<init>()V`	0%	15922 (2.9%)	3417 (13.3%)
`TokenProcessor.action(LExpansion;)V`	0%	6008 (2.8%)	2553 (30.2%)
`RunTimeNfaState.Move(CLjava/util/Vector;)I`	1%	291042 (0.6%)	2339 (0.5%)
_209_db			
`Database.shell_sort(I)V`	20%	135319325 (10.5%)	51963973 (31.7%)
`Entry.equals(Ljava/lang/Object;)Z`	1.1%	3255051 (4.7%)	1704774 (37.5%)
`Database.set_index()V`	2%	3917030 (3.1%)	1375240 (29.3%)
`java.lang.Math.exp(D)D`	0%	76117 (15.2%)	20428 (21.4%)
`Database.read_db(Ljava/lang/String;)V`	0.3%	36671 (0.2%)	9152 (13.2%)
_227_mtrt			
`OctNode.FindTreeNode(LPoint;)LOctNode;`	2.8%	31430193 (14.2%)	518924 (1.5%)
`OctNode.Intersect(LRay;LPoint;F)LOctNode;`	4%	2387161 (0.7%)	167294 (6.3%)
`PolyTypeObj.Intersect(LRay;LIntersectPt;)Z`	0.9%	1683485 (2.2%)	163874 (8.9%)
`Face.GetVert(I)LPoint;`	2.9%	6756582 (2.9%)	82909 (1.1%)
`TriangleObj.Check(LRay;LIntersectPt;)Z`	0.4%	893340 (2.6%)	68419 (7%)
_202_jess			
`jess.Node2.appendToken(Ljess/Token;Ljess/Token;)Ljess/Token;`	0.9%	6031682 (7.8%)	415968 (5.3%)
`jess.Value.<init>(DI)V`	0.2%	802707 (5.4%)	175974 (16.3%)
`jess.RU.getAtom(I)Ljava/lang/String;`	0.4%	547909 (1.7%)	79528 (11.1%)
`jess.Node2.findInMemory(Ljess/TokenVector;Ljess/Token;)Ljess/Token;G`	0.3%	3256222 (12.6%)	26655 (0.6%)
`jess.Value.<init>(II)V`	0%	120408 (8.2%)	20855 (13.2%)
_222_mpegaudio			
`ub.(Lg;)Z`	0.6%	633310 (0.7%)	1876 (0.2%)
`ab. G(I)V`	0.4%	109056 (0.2%)	660 (0.4%)
`d.I([III[FI)V`	0.9%	2800721 (2.5%)	634 (0%)
`q.l([SI)I`	12.7%	5665501 (0.3%)	547 (0%)
`kb. A()V`	0%	74273 (4.6%)	458 (0.4%)

Table 2. The top 5 methods for each of the benchmarks sorted by the number of L2 cache misses.

Method	Accesses	DL1 misses	DL2 misses
_201_compress			
`[B`	24.3%	13348183 (1.6%)	1039633 (6.9%)
`[I`	9.8%	100511389 (29.2%)	58637 (0.1%)
`[S`	5.3%	42087558 (22.8%)	56860 (0.1%)
_213_javac			
`LInstruction;)I`	0.5%	1825299 (7.7%)	82839 (3.1%)
`LFieldExpression;)I`	0.2%	287260 (3.9%)	52025 (13.2%)
`LIdentifierExpression;`	0.2%	359046 (4.5%)	46310 (9.5%)
_228_jack			
`LToken;`	0%	96932 (3.7%)	19864 (12.3%)
`[I`	1.5%	333571 (0.4%)	4007 (0.8%)
`[J`	0.1%	19210 (0.2%)	2529 (8%)
_209_db			
`Ljava/util/Vector;`	4.5%	36870413 (11.4%)	17417400 (38.6%)
`[Ljava/lang/Object;`	2.1%	24912124 (16.3%)	12237002 (41.4%)
`[C`	3.5%	23546444 (9.3%)	12051169 (42.4%)
_227_mtrt			
`LVector;`	1.2%	3558819 (3.5%)	430804 (11%)
`LPoint;`	2.3%	14922491 (7.8%)	351819 (2.1%)
`[LPoint;`	0.8%	10892416 (16.3%)	118231 (1%)
_202_jess			
`[Ljess/ValueVector;`	1.2%	5438243 (5%)	314799 (4.4%)
`Ljess/Value;`	0.8%	3733484 (5.3%)	216368 (4.5%)
`Ljava/lang/Integer;`	0.1%	239721 (2.7%)	48924 (15.3%)
_222_mpegaudio			
`[S`	0.6%	2474016 (2.8%)	1971 (0.1%)
`[F`	12.8%	11833257 (0.6%)	1814 (0%)
`[B`	12.9%	1344543 (0.1%)	1299 (0.1%)

Table 3. The top 3 objects types for each of the benchmarks sorted by the number of L2 cache misses.

Source code	DL1 accesses	DL1 misses	DL2 accesses	DL2 misses
`void shell_sort(int fn) {`				
`  int i, j, n, gap;`				
`  String s1, s2;`				
`  Entry e;`				
`  if (index == null) set_index();`	67	0 (0%)	0	0 (0%)
`  n = index.length;`	134	1 (0%)	1	0 (0%)
`  for (gap = n/2; gap > 0; gap/=2)`	938	0 (0%)	0	(0%)
`    for (i = gap; i < n; i++)`	12276499	910 (0%)	1083	3 (0%)
`      for (j = i-gap; j >=0; j-=gap) {`	23064743	8179 (0%)	9615	33 (0%)
`        s1 = (String)index[j].items.elementAt(fn);`	157553557	29772665 (19%)	36551726	6095594 (17%)
`        s2 = (String)index[j+gap].items.elementAt(fn);`	157553557	24036992 (15%)	29456752	15581062 (53%)
`        if (s1.compareTo(s2) <= 0) break;`	45015302	128 (0%)	153	1 (0%)
`        e = index[j];`	32322537	219 (0%)	228	0 (0%)
`        index[j] = index[j+gap];`	75419253	2654 (0%)	3228	811 (25%)
`        index[j+gap] = e;`	43096716	0 (0%)	0	0 (0%)
`      }`				
`  fnum = fn;`	67	61 (91%)	73	61 (84%)
`}`				

Table 4. The `shell_sort` method from db annotated with cache miss information. The number of L1 and L2 misses here differ from the numbers given Table 2; the reason is that the numbers in this table were obtained using the baseline compiler whereas the numbers in Table 2 were obtained using the adaptive compiler.

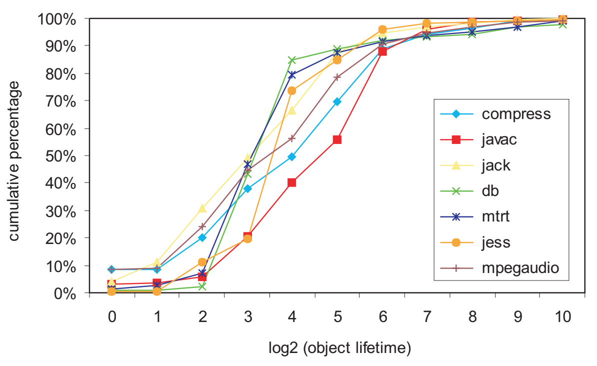

Figure 7. Cumulative object lifetime distribution per benchmark.

on a method call, for knowing the object's type, for knowing the array's length, etc. Also, accesses to objects in native methods in the VM or Java standard libraries need to be hand-instrumented. Tracking all these object references is hard to implement in a virtual machine. In addition, it is very time consuming and error-prone. Measuring the object lifetime within DJ on the other hand is very easy to do and in addition, it is very accurate because it allows for tracking *all* references to a given object. In a VAOMI specification, an object's lifetime can be computed and stored using the per-object `void **userdata` parameter that is available in VAOMI, see section 3.3. As such, computing object lifetimes is very straightforward to do in VAOMI—no more than 50 lines of code.

Figure 7 shows the lifetime distribution for the various benchmarks computed using the DJ system. The X axis on these graphs is given on a log_2 scale; the Y axis shows the cumulative percentage objects in the given lifetime bucket. We observe that the object lifetimes are fairly small in general, *i.e.*, most objects are short-lived objects. For most benchmarks, the object lifetime typically is smaller than 16 memory accesses between the creation of an object and its last use. Some benchmark have a relatively larger object lifetime, see for example javac, compress and mpegaudio, however the object lifetime is still very small in absolute terms, *i.e.*, the object lifetime is rarely more than 64 memory accesses.

6. Related work

6.1 Instrumentation and profiling

A large body of work exist on instrumentation. A number of static instrumentation tools have been proposed such as ATOM [15] and EEL [7]. Static instrumentation cannot be used for analyzing Java applications because it cannot deal with dynamically generated code. Dynamic instrumentation on the other hand does not have that limitation. Well known examples of dynamic binary instrumentation frameworks are Valgrind [11], PIN [8] and DIOTA [10, 9]. In case self-modifying code is supported within the dynamic instrumentation tool, it can be used for vertically profiling Java applications.

One specific tool within the Valgrind's tool set is cachegrind which is a cache profiler that provides limited vertical profiling capabilities. Cachegrind is able to map cache miss rates to lines of source code and methods written in imperative programming languages. However, cachegrind is unable to vertically profile Java applications, nor is it capable of mapping cache miss rates to objects or object types. In addition, the DJ system is much more flexible and allows for a broader use than just cache profiling. In fact, VAOMI allows for building customized vertical memory profiling tools.

6.2 Vertically profiling Java applications

Some recent work focused on vertical profiling of Java applictions. Hauswirth *et al.* [4] presented a vertical profiling approach that correlates hardware performance counter values to manually inserted software monitors in order to keep track of the program's execution across all layers. The low level and high level information is collected at a fairly coarse granularity, *i.e.*, hardware performance counter values and software monitor values are measured once per time slice. There are two important limitations with this approach. First, aligning traces is challenging and caution is required in order not to get out of sync [3]. Our framework does not require explicit alignment. Second, the granularity is very coarse-grained—one performance number per time slice. This allows for analyzing coarse-grained performance variations but does not allow for analyzing fine-grained effects as we target.

Georges *et al*. [2] also provided a limited form of vertical profiling by linking microprocessor-level metrics obtained from hardware performance counters to method-level phases in Java. This allows for analyzing Java applications at a finer granularity than the vertical profiling approach by Hauswirth *et al*. [], however, the granularity is still much more coarse-grained than the granularity that we can achieve in DJ.

The commercially available tool VTune from Intel also allows for profiling Java applications. The VTune tool samples hardware performance counters to profile an application and to annotate source code with cache miss rate information. However, given the fact that VTune relies on sampling it is questionable whether this allows for fine-grained profiling information with little overhead and perturbation of the results.

Note that all of these vertical profiling approaches rely on hardware performance counters which limits the scope of these techniques to evaluating performance on real hardware. Vertical profiling through dynamic binary instrumentation on the other hand, allows for developing customized vertical profiling tools.

7. Summary and future work

Virtualization complicates the understanding of programs being executed on top of the virtualization software. In this paper, we proposed vertical profiling through dynamic binary instrumentation for analyzing Java applications that run on a virtual machine. We proposed a callback mechanism for communicating high level information (related to objects, methods, etc.) from the virtual machine to the dynamic binary instrumentor; this enables the instrumentor to build a vertical profile. The important benefit is that only a few changes need to be made to the VM for enabling vertical profiling. In addition, we also proposed aspect-oriented instrumentation and the VAOMI language for expressing vertical instrumentation specifications in a natural and convenient manner. Finally, we demonstrated the DJ vertical profiling environment that is built around Jikes RVM and DIOTA. In conjunction with the VAOMI language, the DJ system allows for building customized vertical instrumentation tools for analyzing memory behavior in Java applications. The overhead of vertical profiling compared to dynamic instrumentation was quantified to vary between 3.8X and 6.8X depending on what needs to be instrumentated. And we demonstrated the applicability of the DJ system for two applications: vertical cache miss profiling and object lifetime computation. The main benefits of vertical profiling through aspect-oriented dynamic binary instrumentation is that profiling is both efficient to implement and in addition is highly accurate. If this paper gets accepted, the DJ system will be made publicly available.

In future work we plan to further extend the DJ system and the VAOMI language in order to enable vertical profiling of all aspects of a Java application. Our current work focused on memory behavior; in future work, we will extend DJ and VAOMI to also profile other program characteristics such as control flow behavior, and instruction-level parallelism. In addition, we also plan to optimize the DJ system in order to reduce the overall slowdown during vertical profiling.

References

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño Virtual Machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[2] A. Georges, D. Buytaert, L. Eeckhout, and K. De Bosschere. Method-level phase behavior in java workloads. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems and Languages (OOPSLA)*, pages 270–287, October 2004.

[3] M. Hauswirth, A. Diwan, P. S. Sweeney, and M. C. Mozer. Automating vertical profiling. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems and Languages (OOPSLA)*, pages 281–296, October 2005.

[4] M. Hauswirth, P. S. Sweeney, A. Diwan, and M. Hind. Vertical profiling: Understanding the behavior of object-oriented applications. In *Proceedings of the 19th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems and Languages (OOPSLA)*, pages 251–269, October 2004.

[5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, pages 327–355, June 2001.

[6] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242, 1997.

[7] J. R. Larus and E. Schnarr. EEL: Machine-independent executable editing. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, pages 291–300, 1995.

[8] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, pages 190–200, June 2005.

[9] J. Maebe and K. De Bosschere. Instrumenting self-modifying code. In *Proceedings of the Fifth International Workshop on Automated Debugging: AADEBUG2003*, pages 103–113, September 2003.

[10] J. Maebe, M. Ronsse, and K. De Bosschere. DIOTA: Dynamic instrumentation, optimization and transformation of applications. In *Proceedings of the 2002 Workshop on Binary Translation (WBT) held in conjunction with the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, September 2002.

[11] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.

[12] S. Reiss and M. Renieris. Languages for dynamic instrumentation. In *Proceedings of the Workshop on Dynamic Analysis (WODA'03)*, May 2003.

[13] R. Shaham, E. K. Kolodner, and M. Sagiv. Heap profiling for space-efficient java. In *Proceedings of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*, pages 104–113, 2001.

[14] O. Spinczyk, A. Gal, and W. Schröder-Preikschat. AspectC++: an aspect-oriented extension to the C++ programming language. In *CRPITS '02: Proceedings of the Fortieth International Conference on Tools Pacific*, pages 53–60, 2002.

[15] A. Srivastava and A. Eustace. ATOM: A system for building customized program analysis tools. Technical Report 94/2, Western Research Lab, Compaq, March 1994.

[16] J. Suzuki and Y. Yamamoto. Extending UML with aspects: Aspect support in the design phase. In *Proceedings of the Workshop on Object-Oriented Technology*, pages 299–300, 1999.

[17] C. Zhang and H.-A. Jacobsen. TinyC^2: Towards building a dynamic weaving aspect language for C. In *Proceedings of the Foundations of Aspect-Oriented Langauges Workshop at AOSD 2003*, March 2003.